BRL-TR-3339

(2)

TECHNICAL REPORT BRL-TR-3339

BRL

REMOTE DATA TRANSFER (RDT): AN INTERPROCESS DATA TRANSFER METHOD FOR DISTRIBUTED ENVIRONMENTS

JERRY A. CLARKE

MAY 1992

U.S. ARMY LABORATORY COMMAND

BALLISTIC RESEARCH LABORATORY
ABERDEEN PROVING GROUND, MARYLAND

NOTICES

Destroy this report when it is no longer needed. DO NOT return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute indorsement of any commercial product.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAY 1992	Progress, 1 May - 30 Sep 91

4. TITLE AND SUBTITLE

Remote Data Transfer (RdT): An Interprocess Data Transfer Method for Distributed Environments

5. FUNDING NUMBERS

C-AHPCRC
DAAL03-89-7C-0088

6. AUTHOR(S)

Jerry A. Clarke

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Ballistic Research Laboratory
ATTN: SL[illegible]BR-DD-T
Aberdeen Proving Ground, MD 21005-5066

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

BRL-TR-3339

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The decomposition of an application into pieces that can be distributed across several platforms requires a mechanism for exchanging data in addition to orchestrating requests and responses. This paper describes a layer of portable software that can be used to distribute an application in an attempt to achieve maximum system performance.

14. SUBJECT TERMS

RPC: Remote Procedure Call; RdT: Remote Data Transfer;
XDR: External Data Representation, computer programs, software

15. NUMBER OF PAGES
57

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

INTENTIONALLY LEFT BLANK.

TABLE OF CONTENTS

INTENTIONALLY LEFT BLANK.

LIST OF FIGURES

INTENTIONALLY LEFT BLANK.

1. INTRODUCTION

In heterogeneous computing environments, with processors of diverse capabilities and resources, it is often beneficial to distribute an application across several platforms. An application might take advantage of the computational power of one machine, data storage and retrieval of another, and the graphical prowess of a third. Practical considerations such as machine load, availability, and physical connection often are just as important when choosing platforms.

It becomes necessary to have a portable method for transferring data between processes executing on separate machines that may be of different architectures. Remote data Transfer (RdT) provides a layer of utilities, written in the "C" programming language, that enable processes to exchange arbitrary data structures without dealing with many of the specifics of the actual transfer. RdT handles the possible incompatibilities of internal data representation between different architectures as well as the mechanics of communicating that data.

RdT is based on the Client-Server model. A Server makes itself available to perform some service on behalf of a Client. The Server may either wait on incoming requests or periodically check for incoming requests while performing some other function. The Client issues a request then waits for either an acknowledgement that the request has been received or the service performed.

RdT consists of three layers: Data Transfer, Message Passing, and Command and Response. The Data Transfer Layer defines the basic data types that can be exchanged and handles the conversion and passing of the data. The Message Passing Layer handles the registration of services, the delivery and notification of requests, and the return of responses. The Command and Response Layer constructs messages from an independent Command structure and disassembles the response into an independent Response structure.

The Command and Response Layer defines two packages of information: a COMMAND_PACKET and a RETURN_PACKET. These packages are passed to the Message Passing Layer which repacks them into structures known as RT_VARs. Then, using Remote

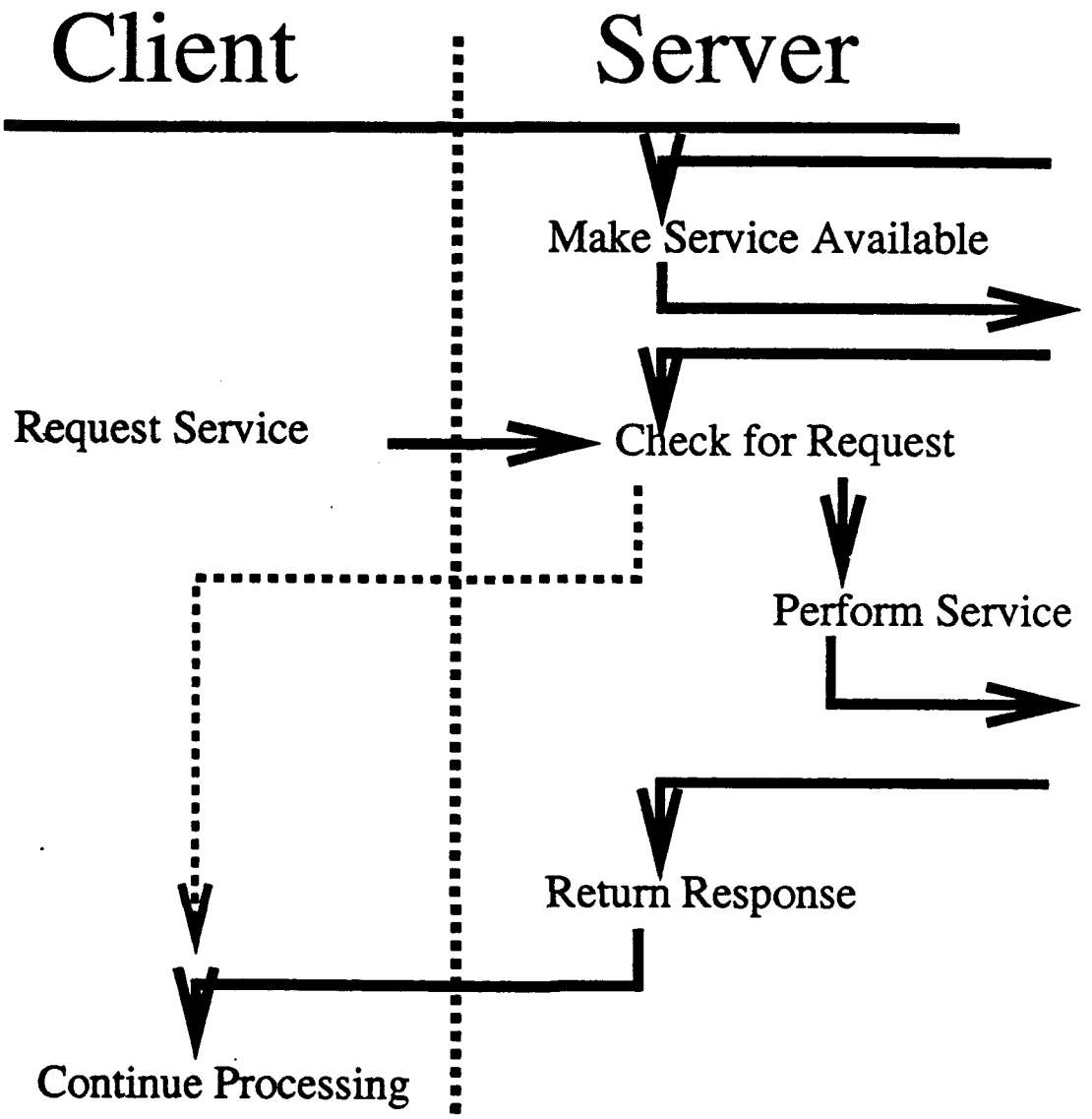

Figure 1. RdT Client-Server Model.

Procedure Calls (RPC) and eXternal Data Representation (XDR), the Data Transfer Layer passes the information between the Client and the Server.

When interfacing with the Command and Response Layer, an application is spared the details of the data transfer. The Command and Response interface can be used locally or between remote processes.

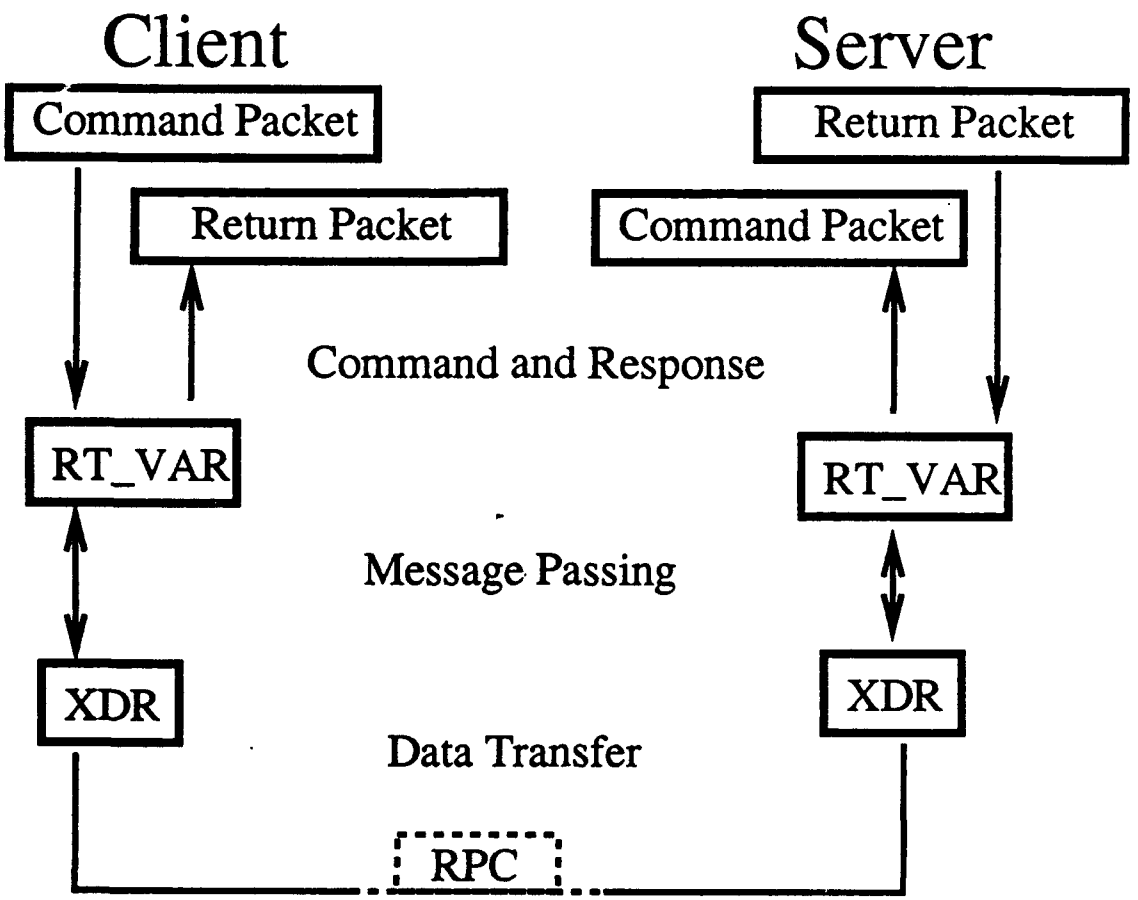

Figure 2. RdT Layers and Packages.

RdT has been tested on the following architectures:

- Sun 3
- Sun 4
- Vax (Ultrix)
- Cray X-MP
- Gould Powernode
- Silicon Graphics
- Alliant
- Convex

RdT should be portable to most UNIX platforms. See appendix for user-callable RdT routines.

2. DATA TRANSFER LAYER

At the lowest layer of RdT, data is transferred between processes using the XDR format developed by Sun Microsystems. The XDR format specifies the bit and byte order of certain data types so that machines that have different internal representations of data can exchange information. To transfer data, RdT uses the following basic XDR data types: *unsigned char* (an 8-bit unsigned character), short (a 16-bit signed integer), long (a 32-bit signed integer), float (a 32-bit signed floating point number), and double (a 64-bit signed floating point number).

Using these basic data types, two additional types are added to the Data Transfer Layer of RdT. The first is a COMPLEX data type that contains two floats defining the real and imaginary parts of a complex number. The other data type is a STRING type that contains a NULL terminated character string and a short integer that defines its length.

There are XDR routines for each of these basic data types that encode and decode each element. These routines can convert the machine specific internal representation of these basic elements to the XDR representation and from XDR to the internal format. When decoding information, these routines must allocate memory for the incoming data if the Server is decoding a command or if the Client is decoding a response.

These basic RdT data types are packaged into a structure called an RT_VAR. An RT_VAR may contain a single data element, or an array of these basic elements. Information in the RT_VAR describes the type of the individual element and the dimensions of the variable. For example, a 10 by 20 array of floats would contain information defining the basic type as a float, define the number of dimensions as 2, and contain an array of length 2 that defines the length of each dimension.

The RT_VAR structure is defined as:

```
typedef struct {
        unsigned char   direction;          /* Server or Client owned */
        unsigned char   type;               /* Basic data type */
        unsigned char   element_size;       /* Byte length of each element */
```

```
unsigned char   n_dim;                  /* Length of dimension array */
long            dim[MAX_ARRAY_DIM]; /* Dimensions */
RT_TYPES        value;                  /* Union of basic data types or */
} RT_VAR                                /* pointer to array */
```

When receiving information, memory space must be allocated to hold the RT_VAR and its data. The element_size and dimension information are used to determine the space requirements while the direction information indicates whether that space can be freed at a later time. For example, if a server process is receiving an array of floats and returns the sum, it must first allocate space for the original array. Once the sum is computed, however, the space can be freed.

These RT_VARS can further be packaged into a STRUCTURE. A STRUCTURE simply contains the number of RT_VARS and an RT_VAR array.

```
typedef struct {
        unsigned long   n_elements;
        RT_VAR          *element[MAX_STRUCT_ELEMENTS];
        } STRUCTURE;
```

By using these data types, processes can pass information as arguments and results. The data can be single elements, arrays, or a combination of the two, i.e., a STRUCTURE. The Data Transfer Layer takes care of unpacking and building these variable types in addition to managing the memory requirements. The valid *types* for an RT_VAR are:

```
#define TYPE_UNDEF      0
#define TYPE_BYTE       1
#define TYPE_SHORT      2
#define TYPE_LONG       3
#define TYPE_FLOAT      4
#define TYPE_DOUBLE     5
#define TYPE_COMPLEX    6
```

```
#define TYPE_STRING      7
#define TYPE_STRUCT      8
```

If the RT_VAR contains more than one basic element, it is ORed with:

```
#define TYPE_ARRAY      0 x 80  /* Compound type */
```

An array of floats would be defined as:

```
rt_var.type = TYPE_FLOAT | TYPE_ARRAY
```

TYPE_STRUCT would contain RT_VARS of type 0 through 7 or arrays of those types. Currently, TYPE_STRUCT cannot contain RT_VARS of TYPE_STRUCT (nested structures) and array of structures are invalid.

Two routines automate the building of RT_VARS: *rtu_make_var* and *rtu_make_structure*.

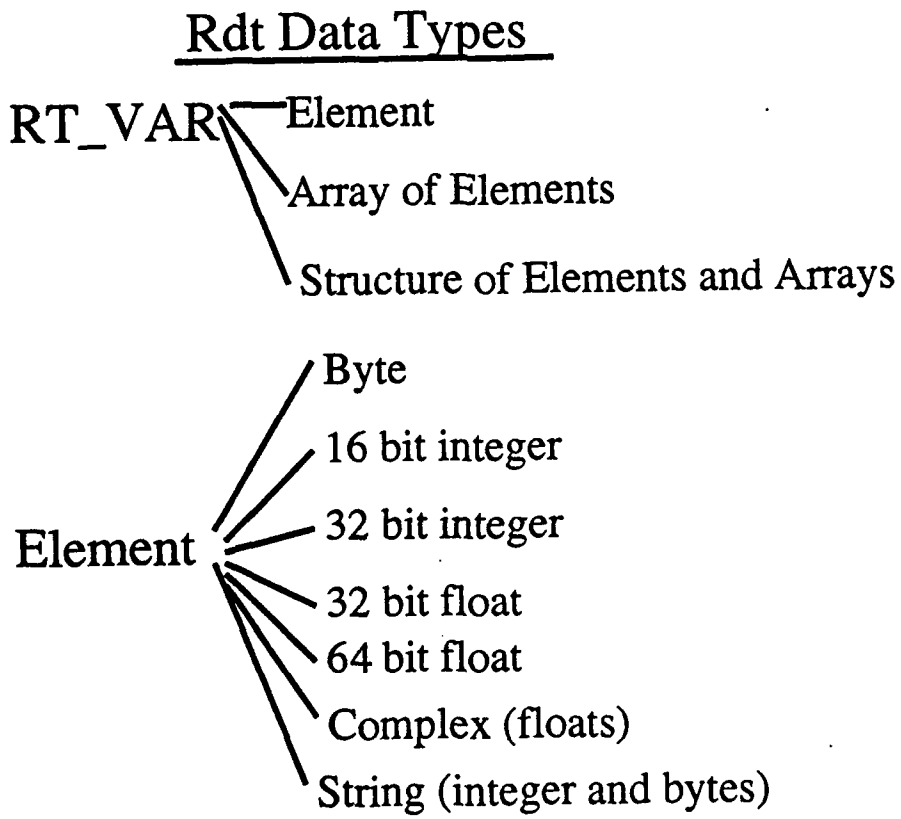

Figure 3. RdT Data Types.

```
int
rtu_make_var(rt_var_ptr, type, length, data)
 RT_VAR          *rt_var_ptr;
 unsigned char   type;
 unsigned long   length;
 char            *data;
```

This routine will build the RT_VAR pointed to by *rt_var_ptr* using the data pointed to by *data*. Notice that *data* must always be a pointer. The *rtu_make_var* only builds single dimension arrays or basic element variables. To build multi-dimensional arrays, the *ndim* and *dimension* RT_VAR structure elements must be modified directly.

To build a STRUCTURE variable, the call *rtu_make_structure* is used. This routine uses a variable argument list so the call would look like:

```
rt_var_ptr = (RT_VAR *)rtu_make_structure(c_structure_pointer,
                                          type,   length,
                                          type,   length,
                                                .
                                                .
                                                .
                                          type,   length,
                                          type,   length,
                                          0);
```

where *type* is an unsigned char and *length* is an unsigned long. So to pass the "C" structure:

```
struct {
      long    array_length;   /* Length of array */
      float   *float_array;   /* Array of length "array_length" */
} my_struct;
```

The call would be:

```
rt_var_ptr = (RT_VAR *)rtu_make_structure(&my_struct,
                        TYPE_LONG,     1,
                        TYPE_FLOAT | TYPE_ARRAY,  mystruct.array_length,
                        0);
```

3. MESSAGE PASSING LAYER

Once the data has been described, it is passed to the next layer of RdT—the Message Passing Layer. This layer is based on the Remote Procedure Call protocol also developed by Sun Microsystems. A Server process registers some service and listens for a request arrive. When one does arrive, the data is decoded from the RT_VARs and passed to some service in the Server. The results of this service are then returned as an RT_VAR.

On the Client side, a request is made for a service to some remote host passing RT_VARs as arguments. The Client makes the request, then waits for a response. If a response is not received in the user defined time limit, the request times out and a TYPE_ERROR is returned. If the Server dies or is unable to decode the arguments, the Client is also notified.

Data transfer is accomplished using Transmission Control Protocol (TCP), the virtual circuit protocol of the Internet protocol family. TCP is used instead of User Datagram Protocol (UDP) since UDP is unreliable and transfers are limited to 8 kB in length. TCP is layered above the Internet Protocol (IP) and provides reliable, flow-controlled, in order, two-way transmission of data. The Server and Client are connected to TCP sockets and transfer their XDR data by reading and writing to these sockets.

3.1 The Server. A Server process makes a call to *rtu_reg* to register a service. This call takes a service number as an argument.

```
int
rtu_reg(service_number)
 int service_number;
```

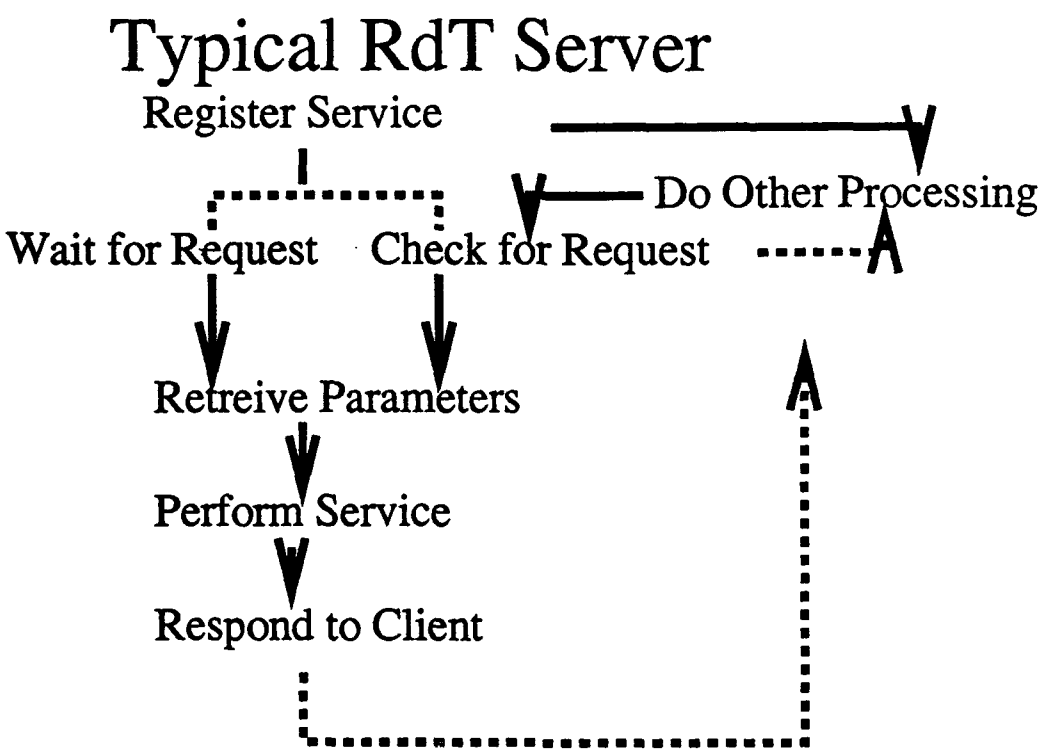

Figure 4. Typical RdT Server.

A RPC service is made unique by a PROGRAM and VERSION number. All RdT services use the same PROGRAM number. They make themselves unique by adding this *service_number* to the RPC VERSION number. When a request for a particular RPC service is received, it is directed to the process that has previously registered that PROGRAM and (VERSION + *service_number*) pair.

The service number is used to distinguish this service from other services on the machine. It is an arbitrary number that the Client will pass to its routines as well. In this way, multiple Servers can exist on the same machine and receive only the calls that are intended for them. The RPC protocol manages the routing of these requests to the proper Server.

The routine *rtu_reg* returns once the service has been registered. To check if there are any incoming requests, the Server makes calls to *rtu_poll*. This routine can be instructed to check for requests and return or to block and wait for an incoming request.

```
int
rtu_poll(user_data, procedure, block_time)
    char    **user_data, **procedure;
    int     block_time;
```

The *user_data*, when returned, will point to a string that contains some user-defined data. This user-defined data is typically some application level authorization (a password) but can be used in any manner the application sees fit.

The *procedure* will also point to a string that was passed by the Client. This information is typically the name of the particular function within a Server that is requested. For example, one server might contain the functions "FFT," "SUM," and "PLOT." A Client would request a particular function from the Server by specifying the name of the function. Again, this is the intended use, but the actual use is determined by the application.

Strings are used for the *user_data* and *procedure* to facilitate the development of simple Servers that call some pre-existing package. The *user_data* can be an access authorization, while the *procedure* can be a command line. In this scenario, RT_VARs would not have to be built or managed.

The *block_time* determines whether *rtu_poll* blocks until a request is received or returns after checking for a request. If *block_time* is negative, *rtu_poll* blocks. Otherwise, the value *block_time* is used as the amount of seconds to wait for an incoming request. If *block_time* is equal to 0, *rtu_poll* checks for a request and returns. In all cases, *rtu_poll* returns a 1 if there is a request or 0 if there is not.

Once *rtu_poll* returns with a request, the Server retrieves the parameters with a call to *rtu_get_par*.

```
char *
rtu_get_par(parameter_number, expected_type)
    int parameter_number;
    unsigned char expected_type;
```

Parameters are numbered starting with zero. The total number of parameters is defined by *RT_NUMARGS*. So the first parameter would be retreived by "rtu_get_par(0, param_type)" and the last would be retrieved by "rtu_get_par(RT_NUMARGS - 1, param_type)."

If the type of the passed in parameter matches the expected type, a pointer is returned that points to that data. For example, if the first expected parameter to a service is a float, the request would be:

```
float    *first_param;

first_param = (float *)rtu_get_par(0, TYPE_FLOAT);
/* For a single float parameter */
/* or */
first_param = (float *)rtu_get_par(0, TYPE_FLOAT | TYPE_ARRAY);
/* For an array of floats */
```

If the *type* does not match, or no such parameter exists (e.g., asking for parameter 10 when only 9 were passed), a NULL pointer is returned. While it would be valid to request the value directly:

```
float first_value;

first_value = *(float *)rtu_get_par(0, TYPE_FLOAT);
```

this is not recommended since *rtu_get_par* can return a NULL, causing the program to fail. It is always wise to make sure that *rtu_get_par* returned a valid parameter.

The Server must always reply to the Client. This reply can be after the requested service has been performed (the usual case) or at some other point in time. Since the Client is blocked when it makes a call to the Server, some returned value must be received before it can continue. The Server responds to the Client with "rtu_reply."

```
int
rtu_reply(service_number, reply_method, reply)
    int    service_number, reply_method;
    RT_VAR *reply;
```

Service number is the same *service_number* used in the call to *rtu_reg*. If *reply_method* is 1, only the reply pointed to by *reply* is returned to the Client. If *reply_method* is 0, then all the parameters that were passed to the Server are returned to the Client. In this manner, the Server could modify some or all of the parameters and pass back the results in the parameters themselves. For large arrays, this is sometimes necessary. Clearly, returning the parameters to the Client requires more communication. In any case, the Server must always reply to the Client with some value, even if that value is only a single byte. There are routines that make responses easy by building the RT_VAR and responding. One such routine is *rtu_smpl_reply*.

```
int
rtu_smpl_reply(type, length, data)
 unsigned char    type;      /* RT_VAR type */
 unsigned long    length;    /* 1 or length of array */
 char             *data;     /* Always a pointer */
```

When the Server is receiving parameters, it must allocate memory to store them. RdT keeps a list of all the memory it has allocated. This list is freed on the next reply. This means that a Server must copy parameters which it wishes to retain before it makes a reply to the Client.

3.2 The Client. The Client requests a service from the server by using *rtu_call*.

```
RT_VAR *
rtu_call(rt_argc, rt_argv, hostname, service_id, procedure, used_data, timeout)
    int          rt_argc;
    RT_VAR       *rt_argv[];
    int          service_id, timeout;
    char         *procedure, *user_data, *hostname;
```

The *rt_argc* defines the number of RT_VARS that are contained in the array *rt_argv*. The *hostname* string contains the name of the machine where the service with *service_id* is located. The *user_data* and *procedure* strings are passed to the Server as described previously.

The *timeout* is the number of seconds to allow for a response. If a response is not received in the allotted time, *rtu_call* returns an error. Errors are returned from rtu_call by the *type* of the returned value being set to TYPE_ERROR (0 x FF).

As with the Server, incoming data (the response) is stored in memory allocated by RdT. This memory is freed on the next call to *rtu_call*. So the Client must copy this response before the next call to *rtu_call* if it wishes to retain the information.

The *rtu_call* blocks until a response is received or an error occurs. To implement non-blocking communication, the Server and Client must cooperate. The Client would first register a service but not issue a *rtu_poll*. Next, the Client would make a call with *rtu_call*, and the Server would respond with a status value indicating that it received the Client's parameters. The Client could then call *rtu_poll* to check on the actual return value to its request. The Server would make the actual response with *rtu_call*. In this situation, both sides are implementing Servers and Clients.

4. COMMAND AND RESPONSE LAYER

The Command and Response Layer (CRL) is implemented above the message passing layer of RdT and based on three basic structures:

```
typedef struct {
        unsigned char       type;        /* Same as basic RdT types */
        int                 num_value;   /* Number of elements in array */
        char                *value;      /* Pointer to data */
        } PARAM_LIST;

typedef struct {
        int                 opcode;      /* Function request */
```

```
    int             num_param;    /* Number of parameters */
    PARAM_LIST      *param;       /* Parameter list */
    } COMMAND_PACKET

typedef struct {
    int             status;       /* Status of operation */
    char            *explain;     /* Error message */
    int             num_return;   /* Number of elements  returned */
    PARAM_LIST      *params;      /* Return params */
    } RETURN_PACKET;
```

CRL is designed to implement the Client-Server model through peer-to-peer communications above the message passing layer. A Client constructs a COMMAND_PACKET, passes it to the Server, and is returned a RETURN_PACKET. The actual transport of the COMMAND_PACKET and RETURN_PACKET is transparent to the Client and Server. CRL could be implemented as shared memory, subroutine calls, or, as in this case, the RdT message layer.

The PARAM_LIST is basically a stripped-down version of an RT_VAR. It only contains the *type* of the data and the number of elements. The *type* is the same as those used in an RT_VAR.

A COMMAND_PACKET contains an opcode which performs the same function as *procedure* in the *rtu_call* routine. This opcode, however, is easier to handle in a "C" switch statement of the Server and is contained within the packet itself. The *num_param* element of the COMMAND_PACKET structure defines the length of the parameter list *param.*

The Client passes a COMMAND_PACKET into CRL and is eventually returned a RETURN_PACKET. A nonnegative status indicates successful completion of the requested service while a negative status indicates an error. Errors are explained in the NULL terminated *explain* element of the packet. The *data* (usually a "C" structure) points to the data that was returned from the Server.

The RETURN_PACKET, which is built by the Server, is unpacked and the *data* returned to the Client. The CRL handles packing and unpacking the data in the RETURN_PACKET.

CRL makes it easy for a Client and Server to exchange arbitrary data structures without concern for the actual delivery mechanism. For example, a Client wants to pass the following structure:

```
struct {
        long        array_length;     /* Length of array */
        float       *float_array;     /* Array of length "array_length" */
        } my_struct;
```

The Client first calls the macro SET_PARAM to put the structure my_struct into a COMMAND_PACKET.

```
COMMAND_PACKET command;

/* SELECT_COMMAND allocates space for the parameters and places some */
/* information like opcode and number of parameters, into the */
/* COMMAND_PACKET.  Usage: */
/* SELECT_COMMAND(command_packet, OPCODE, number_of_parameters) */
SELECT_COMMAND(command, 999, 1);

/* SET_PARAM(command_packet, which_param, type, num_elements, data) */
SET_PARAM(command, 0, TYPE_STRUCT, 1,  rtu_make_structure(&my_struct,
                                TYPE_LONG,  1,
                                TYPE_FLOAT | TYPE_ARRAY,  mystruct.array_length,
                                0);
```

SET_PARAM builds a COMMAND_PACKET for the Command and Response Layer. The opcode is an arbitrary number, agreed upon by the Client and Server to request a particular function within the Server. In response to the request, the Client expects some predefined data

structure. For example, an opcode 999 requested the Server to sum and reverse the array in my_struct. The Server would return some structure RETVAL defined as:

```
typedef struct {
        float       sum;       /* Sum of passed array */
        float       *reverse;  /* Reversed array */
        } RETVAL;

RETVAL *return_val;
```

The actual call that the Client would make to the Server would be:

```
return_val = (RETVAL *)rtu_send_cmd(hostname, service_number, &command);
```

The routine *rtu_send_cmd* calls the Server on machine *hostname* and passes it the COMMAND_PACKET (which, incidently, it passes as a RT_VAR of TYPE_STRUCT). The Server accesses the COMMAND_PACKET, calculates the sum of the array, reverses the array into a return array, and finally responds with a RETURN_PACKET. The *data* element of this packet is returned to the Client from *rtu_send_cmd*. If an error occurred, *rtu_send_cmd* returns a NULL pointer. The Client does not directly unpack the RETURN_PACKET.

The Server uses *rtu_get_cmd* to wait for an incoming COMMAND_PACKET. For example:

```
COMMAND_PACKET *command;
command = (COMMAND_PACKET *)rtu_get_cmd(service_number);
```

Where *service_number* is the same as *service_number* in the call to *rtu_reg*. In implementations other than a RdT message passing layer, this service number could be some other relevant identification such as a shared memory segment.

The Server can now directly access the parameters in the COMMAND_PACKET. For example, if the first expected parameter were "mystruct," the Server would access it with:

```
mystruct_ptr = (MYSTRUCT *)command->param[0].value;
```

In this example, the routine rtu_get_cmd has unpacked the incoming RT_VAR of TYPE_STRUCT into a COMMAND_PACKET. The Server accesses each parameter directly. Once the service has been performed, the Server constructs a RETURN_PACKET and returns it to the Client with a call to rtu_return:

```
rtu_return(packet)
    RETURN_PACKET *packet;
```

The RETURN_PACKET is returned to the Client, and the routine *rt_send_cmd* returns a pointer to this data back to the Client.

The CRL provides a mechanism to implement the Client-Server model which insulates the upper level application from the actual message passing. In this way, a Client-Server application can focus on issues concerning the application with minimal concern for the actual communication mechanism.

5. AN APPLICATION

Many Computational Fluid Dynamics (CFD) codes that execute on supercomputers deal with enormous amounts of data. These codes typically use a grid or set of grids to define discrete points in some computational domain and calculate different properties for these grids in small timesteps. These grids, however, can contain millions of grid points. The solutions can contain many different values for each grid point. There is simply too much data for a smaller machine to handle all at once.

It is desirable, however, to visualize the results of these codes on a much smaller machine such as a workstation. This not only frees the supercomputer for computation but provides many more options for viewing the data since there are a variety of visualization packages that run on workstations.

This use of workstations is only viable, however, if the grid can be accessed in pieces. By visualizing the grid and solution in stages, the entire problem can be viewed by assembling the stages at the end.

To accomplish this grid slicing, the Client-Server model is a good choice. A Server process on the supercomputer responds to requests for a specific slice of the entire grid. The Client process on the workstations makes requests for manageable pieces of the grid. In addition, if these grids on the supercomputer are stored in files, they can remain in the host binary format and do not require the conversion and transmission to the lower end machine.

RTU_SERVER is an example of just such a grid server. It is implemented on the CRL of RdT and supports opcodes that include: open file, close file, read grid (slice), read solution (slice), and check speed. The Client receives information from the Server in a predetermined data structure, GRID_INFO_STRUCT, that contains grid size information as well as the data.

The Server waits for an incoming COMMAND_PACKET. The Client first requests the Server to open a file that is in one of three formats: binary, formatted ASCII, or FORTRAN 77 unformatted. The Server opens the file and returns to the Client the grid size information involved. The Client can then request certain sections of grid (or solution) and eventually request that the file be closed.

For example, suppose we were dealing with a grid big_grid(i, j, k) where i = 64, j = 32, and k = 64, and is stored in a FORTRAN 77 unformatted file called big_grid.dat. To access this grid, the Client makes the call:

```
/* SELECT_COMMAND(command_packet, OPCODE, number_of_parameters) */
SELECT_COMMAND(command, RTU_OPEN, 2);
/* SET_PARAM(command_packet, parameter_number, type, length, data) */
filename = "big_grid.dat";
SET_PARAM(command, 0, TYPE_STRING, 1, filename);
file_type = F77_UNFORMATTED;
SET_PARAM(command, 1, TYPE_LONG, 1, &file_type);
```

```
if((grid_info = (GRID_INFO_STRUCT *)rtu_send_cmd(hostname,
service_number, &command)) == NULL){
        fprintf(stderr, "Unable to open file %s on %s\n," filename, hostname);
        exit(1);
        }
```

The Client then selects which slice of the grid is desired, stores that information back in the grid_info structure, and sends the structure as a parameter to the "read grid" function of the Server:

```
grid_info->which_grid = 0;          /* A grid file can contain multiple grids */
grid_info->num_planes = 2;
grid_info->which_plane = KPLANE;
grid_info->planes[0] = 2;
grid_info->planes[1] = 16;

SELECT_COMMAND(command, RTU_READ_GRID, 1);

SET_PARAM(command, 0, TYPE_STRUCT, 1,
            rtu_make_structure(grid_info, GRID_INFO_DESC(grid_info)));
/* GRID_INFO_DESC() is a macro that constructs the proper */
/* parameter list for rtu_make_structure() from the structure */
/* in grid_info. */

if((grid = (GRID_DATA_STRUCT *)rtu_send_cmd(hostname,
service_number, &command)) == NULL){
        fprintf(stderr, "Error reading grid from %s\n", hostname);
        exit(1);
        }
```

In similar manner, the Client reads the same slices of the solution file from the Server. By requesting the grid and solution in manageable pieces, the low end machine is able to deal with a huge amount of data. In addition, there is no need to convert the FORTRAN 77 unformatted

files to another format and/or transfer them to the low end machine with possibly modest disk resource.

This grid server (Figure 5) is currently used in a distributed visualization environment. The computation on the supercomputer can proceed in parallel with the visualization which has been off-loaded to another machine. The visualization platform can be chosen not only based on processing power or graphics hardware but on practical considerations such as machine load physical connection and availability.

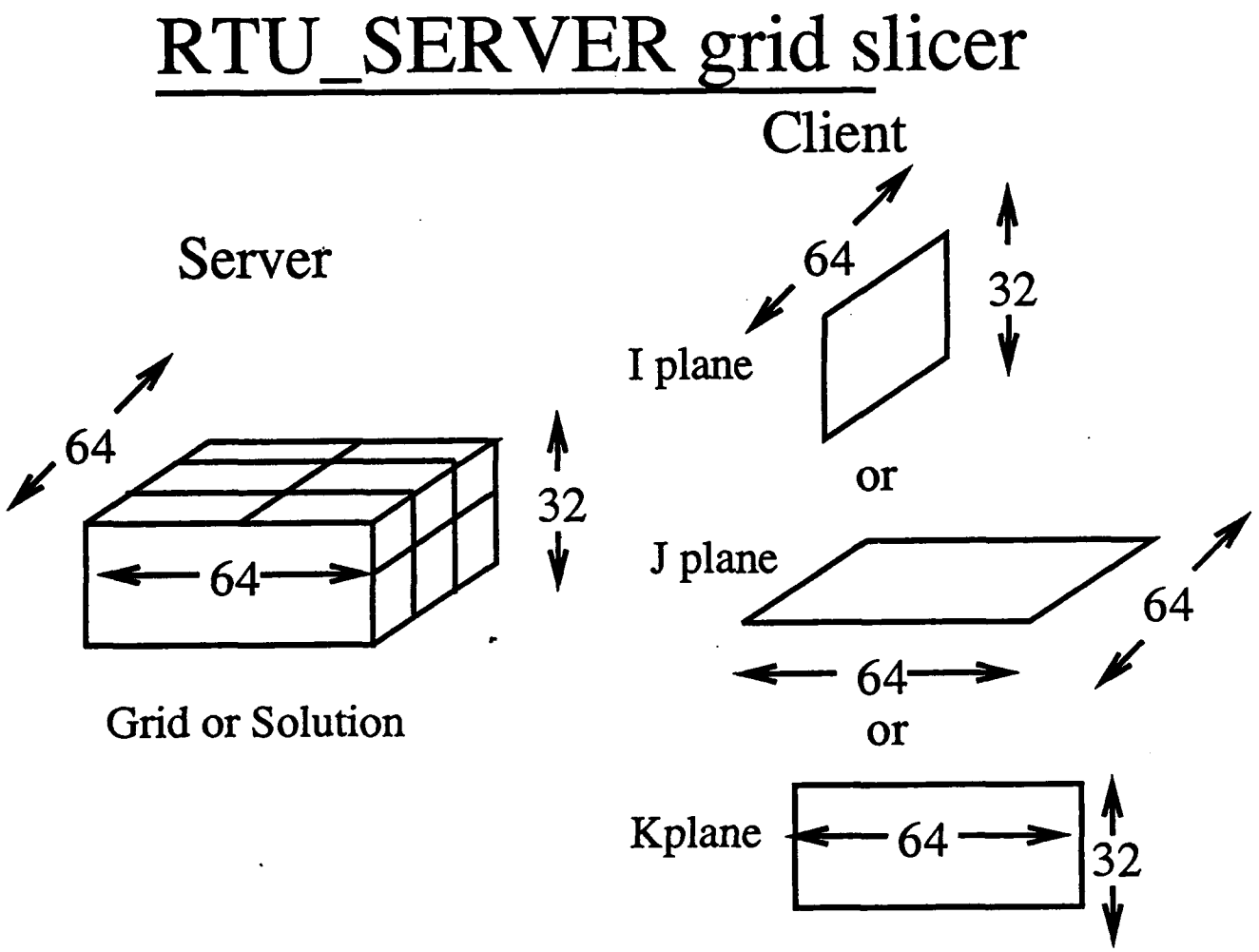

Figure 5. RTU_Server Grid Slicer.

6. CONCLUSIONS

RdT provides layered software to provide a data transfer method in distributed environments. While RdT minimizes some of the housekeeping, it naturally introduces some overhead. The actual wall clock transfer time varies with machine load on the local and remote machines as well as the network traffic at a given time.

Typical transfer times between two Sun 4 machines on a reasonably loaded network result in transfers of 10,000 floats in about 0.1 s or in the neighborhood of 1/3 to 1/2 MB/s. Raw TCP/IP packets result in transfer of less than 1 MB/s. These times are not directly comparable, but are useful when designing a distributed application where such transfer times may or may not be significant. It is important to mention that transfer times do not vary linearly. The RPC/XDR overhead to transfer 10 numbers is similar to that of transferring 1,000. So transferring larger arrays produces better than average throughput than the transfer of small arrays. This is generally true until the size of the array and RdT overhead approach the size of physical memory, where the allocation of memory and swapping introduce significant overhead.

RdT has been used to distribute the separate functions of an application across several platforms. Each function can then take advantage of a particular architecture or set of resources. To take full advantage of a distributed environment, however, RdT could use some additional flexibility. Future enhancements include additional data transport mechanisms such as shared memory. The message passing layer can then choose the most appropriate method for data transport. For example, if data is being requested from the local host, shared memory or direct subroutine calls would be selected to implement the Command and Response Layer. In addition, the message passing layer will be made more flexible to help implement non-blocking and intermediate message passing where a message is passed through several Servers to reach its destination or is broadcast to many targets.

INTENTIONALLY LEFT BLANK.

APPENDIX:

RdT USER ROUTINES

INTENTIONALLY LEFT BLANK.

NAME

rtu_calloc - Allocate memory

SYNTAX

```
#include <rtrpc.h>

char *rtu_calloc(num_elements, element_size)
int num_elements;
int element_size;
```

PARAMETERS

num_elements is the number of element for which to allocate memory.

element_size is the number of bytes in each element.

DESCRIPTION

Rtu_calloc will allocate num_elements * element_size bytes of memory; possibly by calling calloc(). A pointer to this space is maintained in an internal list. The entire internal list of allocated memory is freed on the next call to rtu_cfree.

SPECIAL CONSIDERATIONS

Rtu_calloc may not always use calloc to allocate memory depending on the architecture.

DIAGNOSTICS

Rtu_calloc returns a pointer to the memory that has been allocated or NULL if there was an error.

SEE ALSO

rtu_poll(3R), rtu_call(3R), rtu_cfree(3R)

NAME

rtu_cfree - free memory

SYNTAX

```
#include <rtrpc.h>
void rtu_cfree()
```

PARAMETERS

none

DESCRIPTION

Rtu_cfree frees all of the memory that has been allocated using rtu_calloc.

SPECIAL CONSIDERATIONS

Rtu_cfree usually calls cfree.

DIAGNOSTICS

Rtu_cfree returns no value.

SEE ALSO

rtu_poll(3R), rtu_call(3R), rtu_calloc(3R)

NAME

rtu_call - call an RdT server

SYNTAX

```
#include <rtrpc.h>
int rtu_call(rt_argc, rt_argv, hostname,
    . . .  service_id, procedure, user_data, timeout);
int rt_argc, service_id, timeout;
RT_VAR *rt_argv[];
char *hostname, *user_data, *procedure;
```

PARAMETERS

Rt_argc defines the number of RT_VAR pointers in the array rt_argv.

rt_argv is an array of pointers to RT_VAR data structures.

hostname is the hostname of the RdT server to be called.

service_id is the number of the service that is requested.

procedure is the name of the particular function inside the requested service.

user_data is a user defined string that is passed to the server. It is usually used as an authentication.

timeout is the number of seconds to allow for a response from the server.

DESCRIPTION

A RT_VAR, defined in rtrpc.h, and usually built by rtu_make_var, is the basic data structure used by RdT. A RT_VAR must be one of the following types

```
#define TYPE_UNDEF      0
#define TYPE_BYTE       1
#define TYPE_SHORT      2
#define TYPE_LONG       3
#define TYPE_FLOAT      4
#define TYPE_DOUBLE     5
#define TYPE_COMPLEX    6
#define TYPE_STRING     7
#define TYPE_STRUCT     8
```

If the parameter is an array of one of these types, it is ORed with:

#define TYPE_ARRAY 0 x 80 /* Compound type */

Rtu_call passes a list of RT_VARs to a RdT server running on the machine hostname that has registered the service_id.

The NULL terminated strings user_data and procedure are also passed to the server. Procedure is intended to identify the function inside the server that is requested. user_data is intended to be used as an authentication agreed upon by the client and server. The actual use of procedure and user_data, however, is entirely implementation specific.

Once the call is made, the client will be blocked for timeout seconds, or until a response is received from the server. A negative time out causes the client to block, while a zero time out returns immediatly.

SPECIAL CONSIDERATIONS

- A Server must always respond to a client since the client is blocked until a reply is received.
- If hostname is NULL, then the local host is used.
- user_data and procedure may also be NULL.

DIAGNOSTICS

Rtu_call always returns a RT_VAR. If an error occurred, the type of the RT_VAR is set to TYPE_ERROR. If successful, the RT_VAR returned is the RT_VAR that was passed by the server to the routine rtu_reply.

SEE ALSO

rtu_reg(3R), rtu_reply(3R), rtu_poll(3R), rtu_get_par(3R), rtu_calloc(3R) rtu_cfree(3R), rtu_smpl_reply(3R)

NAME

rtu_get_cmd - Retrieve a RdT COMMAND PACKET

SYNTAX

```
#include <rtu_util.h>
COMMAND_PACKET *rtu_get_cmd(service_number)
int  service_number;
```

PARAMETERS

Service_number identifies the service that is being performed by the server on behalf of the client.

DESCRIPTION

Rtu_get_cmd implements the server side of the Command and Response layer (CRL) of RdT. CRL is based upon three basic data structures:

```
typedef struct {
   unsigned char     type;     /* Basic RdT data type */
   int               num_values; /* Number of elements */
   char              *data;      /* Pointer to Data */
   } PARAM_LIST;

typedef struct {
   int               opcode; /* Function request */
   int               num_param; /* Number of parameters */
   PARAM_LIST   *params; /* Pointers to parameters */
   } COMMAND_PACKET;

typedef struct {
   int               status; /* Status of command */
   char              *explain; /* Explanation of Bad status */
   int               num_return; /* Number of returned variables */
   PARAM_LIST   *params; /* Pointers to return parameters */
   } RETURN_PACKET;
```

Rtu_get_cmd retrieves a COMMAND_PACKET from a client program. Some function is performed by the server and a RETURN_PACKET is passed back to the client.

The macro SET_PARAM() can be used to pack data into a RETURN_PACKET: SET_PARAM(RETURN_PACKET packet, int which_param, unsigned char type, int array_size, char *data)

SPECIAL CONSIDERATIONS

- A Server must always respond to a client since the client is blocked until a reply is received.

DIAGNOSTICS

Rtu_get_cmd returns NULL on an error after printing an explanatory message to stderr. If successful, rtu_get_cmd returns a pointer to the COMMAND_PACKET that was sent by the client to the server.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_poll(3R), rtu_get_par(3R), rtu_calloc(3R) rtu_cfree(3R), rtu_smpl_reply(3R), rtu_s_cmd(3R), rtu_reply(3R)

NAME

rtu_get_par - Retrieve RdT parameters

SYNTAX

```
#include <rtrpc.h>
char * rtu_get_par(parameter_number, expected_type)
intparameter_number;
unsigned char expected_type;
```

PARAMETERS

parameter_number specifies which parameter (numbered from 0) is to be retrieved.

expected_type specifies the type of the parameter that is expected. Expected_type is one of the rt_var types defined in rtrpc.h:

```
#define TYPE_UNDEF      0
#define TYPE_BYTE       1
#define TYPE_SHORT      2
#define TYPE_LONG       3
#define TYPE_FLOAT      4
#define TYPE_DOUBLE     5
#define TYPE_COMPLEX    6
#define TYPE_STRING     7
#define TYPE_STRUCT     8
```

If the parameter is an array of one of these types, it is ORed with:

```
#define TYPE_ARRAY      0 x 80  /* Compound type */
```

DESCRIPTION

Rtu_get_par returns a pointer to the data of the requested parameter. The return pointer from rtu_get_par must be cast to the proper type. For example, if the first expected parameter was an array of floats, the call would be:

```
float *first_param;
first_param = (float *)rtu_get_par(0,
        TYPE_FLOAT | TYPE_ARRAY);
```

If rtu_get_par was successful in retrieving the first parameter, the data is pointed to by the pointer first_param.

SPECIAL CONSIDERATIONS

- The expected type of the parameter must match exactly.
- The data pointed to by rtu_get_par has been allocated using rtu_calloc. That storage space will be freed on the next call to rtu_reply. So the data must be copied before a response is made or the data will be corrupted.

DIAGNOSTICS

Rtu_get_par returns NULL if the expected_type does not match the parameter or if there are less parameters than the number that was requested with parameter_number. Otherwise, rtu_get_par returns a pointer to the data.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_poll(3R), rtu_reply(3R), rtu_calloc(3R) rtu_cfree(3R)

NAME

rtu_make_structure - build an RT_VAR

SYNTAX

```
#include <rtrpc.h>
int rtu_make_structure(c_structure_ptr,
                type, length,
                    .
                    .
                    .
                    0)

char *c_structure_ptr;
unsigned char type;
unsigned long length;
```

PARAMETERS

C_structure_ptr is a pointer to an a 'C' structure.

Type is the RT_VAR type of the data of the RT_VAR to be built.

Length is the number of elements in the array pointed to by associated 'C' structure element or 1 if the element is a scaler.

DESCRIPTION

Rtu_make_structure builds an RT_VAR of type TYPE_STRUCT from the data pointed to by c_structure_ptr.

Rtu_make_structure uses a variable argument list terminated by a 0. This allows an arbitrary sized 'C' structure to be passed and defined. There must be one type, length pair for each element of the structure.

Type must be one of the rt_var types defined in rtrpc.h:

```
#define TYPE_UNDEF      0
#define TYPE_BYTE       1
#define TYPE_SHORT      2
#define TYPE_LONG       3
#define TYPE_FLOAT      4
#define TYPE_DOUBLE     5
#define TYPE_COMPLEX    6
#define TYPE_STRING     7
#define TYPE_STRUCT     8
```

If the parameter is an array of one of these types, it is ORed with:

```
#define TYPE_ARRAY      0 x 80  /* Compound type */
```

SPECIAL CONSIDERATIONS

- Rtu_make_structure allocates the RT_VAR that is returned with rtu_calloc. This will be freed on the next call to rtu_call or rtu_reply.

DIAGNOSTICS

Rtu_make_structure returns an pointer to a RT_VAR or NULL if the was an error.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_poll(3R), rtu_get_par(3R), rtu_calloc(3R) rtu_cfree(3R), rtu_reply(3R), rtu_smpl_reply(3R)

NAME

rtu_make_var - build a RT_VAR

SYNTAX

```
#include <rtrpc.h>
int  rtu_make_var(rt_var_ptr, type,
                  length, data_pointer)
RT_VAR *rt_var_ptr;
unsigned char type;
unsigned long length;
char *data_pointer;
```

PARAMETERS

rt_var_ptr is a pointer to a RT_VAR data structure defined in rtrpc.h.

Type is the RT_VAR type of the data of the RT_VAR to be built.

Length is the number of elements in the array pointed to by data_pointer, or 1 if the data is a scaler.

DESCRIPTION

Rtu_make_var builds a RT_VAR from the supplied information and places the variable in the RT_VAR pointed to by rt_var_ptr. Type must be one of the rt_var types defined in rtrpc.h:

```
#define TYPE_UNDEF      0
#define TYPE_BYTE       1
#define TYPE_SHORT      2
#define TYPE_LONG       3
#define TYPE_FLOAT      4
#define TYPE_DOUBLE     5
#define TYPE_COMPLEX    6
#define TYPE_STRING     7
#define TYPE_STRUCT     8
```

If the parameter is an array of one of these types, it is ORed with:

```
#define TYPE_ARRAY      0 x 80  /* Compound type */
```

SPECIAL CONSIDERATIONS

- rt_var_ptr must point to a RT_VAR; the RT_VAR is not allocated by rtu_make_var.

DIAGNOSTICS

Rtu_make_var returns 1 on success 0 on failure.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_poll(3R), rtu_get_par(3R), rtu_calloc(3R) rtu_cfree(3R), rtu_reply(3R), rtu_smpl_reply(3R)

NAME

rtu_poll - listen for a RdT service request

SYNTAX

```
#include <rtrpc.h>
int rtu_poll(user_data, procedure, block_time)
char **user_data;
char **procedure;
int  block_time;
```

PARAMETERS

user_data points to a character string of user specific data. This is usually used as some type of access authorization but can be used to pass any character string.

procedure points to a character string that usually specifies the name of the procedure within the requested service that the client program desires. Again this is the intended use, but the client is free to pass any information in this string.

block_time specifies the type and length of blocking. If block_time is negative, rtu_poll does not return until a service request is received from a client. If block_time is zero, rtu_poll checks for an incoming service request and returns immediately. Any other value is taken as the amount of seconds to block for an incoming service request before returning.

DESCRIPTION

Rtu_poll listens for a request of a RdT service that has been previously registered with rtu_reg. When a request is received, the user_data and procedure values are read from the client and stored in a storage area that is pointed to by user_data and procedure. When a valid request has been received, the parameters passed from the client are retrieved with rtu_get_par. The server generally checks the data in user_data to assure that the client is authorized to make the request, then passes all parameters to the particular function specified by procedure.

SPECIAL CONSIDERATIONS

- rtu_poll must be called after rtu_reg.

DIAGNOSTICS

Rtu_poll returns 1 if a valid request has been received or 0 if no service request has been received.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_get_par(3R), rtu_reply(3R)

NAME

rtu_reg - register a RdT service

SYNTAX

```
#include <rtrpc.h>
int  rtu_reg(service_number)
int service_number
```

PARAMETERS

service_number a unique number identifying the service

DESCRIPTION

Rtu_reg registers a RdT service uniquely identified by service_number. A client process requesting service_number will be served by this service.

SPECIAL CONSIDERATIONS

- Rtu_reg overrides an previous call. Any service that has been previously registered with the same service_number is unreachable.

DIAGNOSTICS

Rtu_reg returns 1 if the service was registers successfully or 0 on an error.

SEE ALSO

rtu_poll(3R), rtu_call(3R)

NAME

rtu_reply - reply to an RdT client

SYNTAX

```
#include <rtrpc.h>
int rtu_reply(service_number, reply_method, reply)
int service_number;
int reply_method;
RT_VAR *reply;
```

PARAMETERS

Service number identifies the service that was specified with rtu req that is being performed on behalf of the client.

Reply method specifies if the passed parameters are returned to the client along with the reply.

reply is the actual reply data sent to the client. reply must be one of the rt var types defined in rtrpc.h:

```
#define TYPE_UNDEF      0
#define TYPE_BYTE       1
#define TYPE_SHORT      2
#define TYPE_LONG       3
#define TYPE_FLOAT      4
#define TYPE_DOUBLE     5
#define TYPE_COMPLEX    6
#define TYPE_STRING     7
#define TYPE_STRUCT     8
```

If the parameter is an array of one of these types, it is ORed with:

```
#define TYPE_ARRAY      0 x 80  /* Compound type */
```

DESCRIPTION

Rtu reply is the most general return routine in the RdT package. For simple replies of scalars and arrays, rtu smpl reply() is usually sufficient.

Rtu reply sends the data specified in reply back to the RdT client in response to a service request. If reply method is RETURN_PARAMS, the parameters passed to the server from rtu get par, are returned to the client. This allows the server to change the data in a parameter and return it to a client without the need to allocate new memory. If reply method is set to REPLY_ONLY no parameters are returned. This is faster than returning all parameters.

SPECIAL CONSIDERATIONS

- A Server must always respond to a client since the client is blocked until a reply is received.

DIAGNOSTICS

Rtu_reply returns 1 on success 0 on failure.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_poll(3R), rtu_get_par(3R), rtu_calloc(3R) rtu_cfree(3R), rtu_smpl_reply(3R)

NAME

rtu_return - respond to a RdT client

SYNTAX

```
#include <rtu_util.h>
int  rtu_return(return_packet)
RETURN_PACKET   *return_packet;
```

PARAMETERS

Return_packet is the data structure that contains the response to the client.

DESCRIPTION

Rtu_return helps implement the server side of the Command and Response Layer (CRL) of RdT. CRL is based upon three basic data structures:

```
typedef struct {
   unsigned char          type;     /* Basic RdT data type */
   int                    num_values; /* Number of elements */
   char                   *data;   /* Pointer to Data */
   } PARAM_LIST;

typedef struct {
   int                    opcode; /* Function request */
   int                    num_param; /* Number of parameters */
   PARAM_LIST             *params; /* Pointers to parameters */
   } COMMAND_PACKET;

typedef struct {
   int                    status; /* Status of command */
   char                   *explain; /* Explanation of Bad status */
   int                    num_return; /* Number of returned variables */
   PARAM_LIST             *params; /* Pointers to return parameters */
   } RETURN_PACKET;
```

Rtu_return returns the RETURN_PACKET to the client in response to some request for service. The macro SET_PARAM() can be used to pack data into a RETURN_PACKET:

```
SET_PARAM(RETURN_PACKET packet, int which_param,
unsigned char type,
int array_size, char *data)
```

SPECIAL CONSIDERATIONS

- A Server must always respond to a client since the client is blocked until a reply is received.

DIAGNOSTICS

Rtu_return returns 1 on success 0 on failure.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_poll(3R), rtu_get_par(3R), rtu_calloc(3R) rtu_cfree(3R), rtu_smpl_reply(3R), rtu_s_cmd(3R), rtu_reply(3R)

NAME

rtu_send_cmd - Send a RdT COMMAND PACKET

SYNTAX

```
#include <rtu_util.h>

char *rtu_send_cmd(hostname, service_number,
      command_packet)
char *hostname;
int   service_number;
COMMAND_PACKET *command_packet;
```

PARAMETERS

Hostname is the name of the host that is executing the Server side of RdT.

Service number identifies the service that was specified with rtu_req that is being performed on behalf of the client.

Command_packet points to a valid COMMAND_PACKET.

DESCRIPTION

Rtu_send_cmd implements the client side of the Command and Response layer (CRL) of RdT. CRL is based upon three basic data structures:

```
typedef struct {
   unsigned char     type;    /* Basic RdT data type */
   int               num_values; /* Number of elements */
   char              *data;/* Pointer to Data */
   } PARAM_LIST;

typedef struct {
   int               opcode; /* Function request */
   int               num_param; /* Number of parameters */
   PARAM_LIST        *params; /* Pointers to parameters */
   } COMMAND_PACKET;

typedef struct {
   int               status; /* Status of command */
   char              *explain; /* Explanation of Bad status */
   int               num_return; /* Number of returned variables */
   PARAM_LIST        *params; /* Pointers to return parameters */
   } RETURN_PACKET;
```

Rtu_send_cmd sends a COMMAND_PACKET to a server program running on hostname that has been registered with service_number. When the server responds with a RETURN_PACKET, a pointer to the data element of the structure is returned as a result of rtu_send_cmd.

The macros SELECT_COMMAND() and SET_PARAM() are usually called to build the COMMAND_PACKET:

```
SELECT_COMMAND(COMMAND_PACKET packet, int opcode,
        int number_of_parameters)

SET_PARAM(COMMAND_PACKET packet, int which_param,
        unsigned char type,
        int array_size, char *data)
```

SPECIAL CONSIDERATIONS

- A Server must always respond to a client since the client is blocked until a reply is received.

DIAGNOSTICS

Rtu_send_cmd returns NULL on an error after printing an explanatory message to stderr. If successful, rtu_send_cmd returns a pointer to the data that was returned as a result from the server.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_poll(3R), rtu_get_par(3R), rtu_calloc(3R) rtu_cfree(3R), rtu_smpl_reply(3R), rtu_get_cmd(3R)

RTU_SMPL_REPLY 3R

NAME

rtu_smpl_reply - reply to a RdT client

SYNTAX

```
#include <rtrpc.h>
 int  rtu_smpl_reply(type, length, data_pointer)
 unsigned char type;
 unsigned long length;
 char *data_pointer;
```

PARAMETERS

Type is the RT_VAR type of the reply data sent to the client. Type must be one of the rt_var types defined in rtrpc.h:

```
#define TYPE_UNDEF      0
#define TYPE_BYTE       1
#define TYPE_SHORT      2
#define TYPE_LONG       3
#define TYPE_FLOAT      4
#define TYPE_DOUBLE     5
#define TYPE_COMPLEX    6
#define TYPE_STRING     7
#define TYPE_STRUCT     8
```

If the parameter is an array of one of these types, it is ORed with:

```
#define TYPE_ARRAY    0 x 80  /* Compound type */
```

Length is the number of elements in the array pointed to by data_pointer, or 1 if the data is a scaler.

DESCRIPTION

Rtu_smpl_reply builds a RT_VAR from the information supplied and passes it to rtu_reply to be passed back to a RdT client. Only the data pointed to by data_pointer is returned to the client; no parameters are returned.

SPECIAL CONSIDERATIONS

• A Server must always respond to a client since the client is blocked until a reply is received.

DIAGNOSTICS

Rtu_reply returns 1 on success 0 on failure.

SEE ALSO

rtu_reg(3R), rtu_call(3R), rtu_poll(3R), rtu_get_par(3R), rtu_calloc(3R) rtu_cfree(3R), rtu_reply(3R)

No. of Copies	Organization
2	Administrator Defense Technical Info Center ATTN: DTIC-DDA Cameron Station Alexandria, VA 22304-6145
1	Commander U.S. Army Materiel Command ATTN: AMCAM 5001 Eisenhower Ave. Alexandria, VA 22333-0001
1	Commander U.S. Army Laboratory Command ATTN: AMSLC-DL 2800 Powder Mill Rd. Adelphi, MD 20783-1145
2	Commander U.S. Army Armament Research, Development, and Engineering Center ATTN: SMCAR-IMI-I Picatinny Arsenal, NJ 07806-5000
2	Commander U.S. Army Armament Research, Development, and Engineering Center ATTN: SMCAR-TDC Picatinny Arsenal, NJ 07806-5000
1	Director Benet Weapons Laboratory U.S. Army Armament Research, Development, and Engineering Center ATTN: SMCAR-CCB-TL Watervliet, NY 12189-4050
(Unclass. only)1	Commander U.S. Army Armament, Munitions, and Chemical Command ATTN: AMSMC-IMF-L Rock Island, IL 61299-5000
1	Director U.S. Army Aviation Research and Technology Activity ATTN: SAVRT-R (Library) M/S 219-3 Ames Research Center Moffett Field, CA 94035-1000
1	Commander U.S. Army Missile Command ATTN: AMSMI-RD-CS-R (DOC) Redstone Arsenal, AL 35898-5010

No. of Copies	Organization
1	Commander U.S. Army Tank-Automotive Command ATTN: ASQNC-TAC-DIT (Technical Information Center) Warren, MI 48397-5000
1	Director U.S. Army TRADOC Analysis Command ATTN: ATRC-WSR White Sands Missile Range, NM 88002-5502
1	Commandant U.S. Army Field Artillery School ATTN: ATSF-CSI Ft. Sill, OK 73503-5000
2	Commandant U.S. Army Infantry School ATTN: ATZB-SC, System Safety Fort Benning, GA 31903-5000
(Class. only)1	Commandant U.S. Army Infantry School ATTN: ATSH-CD (Security Mgr.) Fort Benning, GA 31905-5660
(Unclass. only)1	Commandant U.S. Army Infantry School ATTN: ATSH-CD-CSO-OR Fort Benning, GA 31905-5660
1	WL/MNOI Eglin AFB, FL 32542-5000
	Aberdeen Proving Ground
2	Dir, USAMSAA ATTN: AMXSY-D AMXSY-MP, H. Cohen
1	Cdr, USATECOM ATTN: AMSTE-TC
3	Cdr, CRDEC, AMCCOM ATTN: SMCCR-RSP-A SMCCR-MU SMCCR-MSI
1	Dir, VLAMO ATTN: AMSLC-VL-D
10	Dir, USABRL ATTN: SLCBR-DD-T

INTENTIONALLY LEFT BLANK.

USER EVALUATION SHEET/CHANGE OF ADDRESS

This laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers below will aid us in our efforts.

1. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

2. How, specifically, is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

3. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, .etc? If so, please elaborate. _____

4. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

BRL Report Number BRL-MR-3340 Division Symbol _____

Check here if desire to be removed from distribution list. ____

Check here for address change. ____

Current address: Organization _____
Address _____

DEPARTMENT OF THE ARMY
Director
U.S. Army Ballistic Research Laboratory
ATTN: SLCBR-DD-T
Aberdeen Proving Ground, MD 21005-5066

OFFICIAL BUSINESS

BUSINESS REPLY MAIL
FIRST CLASS PERMIT No 0001, APG, MD

Postage will be paid by addressee.

Director
U.S. Army Ballistic Research Laboratory
ATTN: SLCBR-DD-T
Aberdeen Proving Ground, MD 21005-5066

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES